

# Reducing False Positives in Runtime Analysis of Deadlocks

S. Bensalem\*

Klaus Havelund†

## Abstract

This paper presents an improvement of a standard algorithm for detecting deadlock potentials in multi-threaded programs, in that it reduces the number of false positives (false warnings). The standard algorithm works as follows. The multi-threaded program under observation is executed, while *lock* and *unlock* events are observed. A graph of locks is built, with edges between locks symbolizing locking orders. Any cycle in the graph signifies a potential for a deadlock. The typical standard example is the group of dining philosophers sharing forks. The algorithm is interesting because it can catch deadlock potentials even though no deadlocks occur in the examined trace, and at the same time it scales very well in contrast to more formal approaches to deadlock detection. The algorithm, however, can yield false positives, as well as false negatives (missed errors). The extension of the algorithm described in this paper reduces the amount of false positives for three particular cases: when a gate lock protects a cycle, when a single thread introduces a cycle, and when the code segments in different threads that cause the cycle can actually not execute in parallel. The paper formalizes a theory for dynamic deadlock detection and compares it to model checking and static analysis techniques. It furthermore describes an implementation for analyzing Java programs and its application to two case studies: a planetary rover and a space craft altitude control system.

## 1 Introduction

Concurrent programming can in some situations give a programmer a flexibility in organizing interacting code modules in a conceptually much simpler way than is possible with sequential programming. It can potentially also speed up a program in case a multi-processor architecture is used. The Java programming language [1] explicitly supports concurrent programming through a selection of concurrency language concepts, such as threads and monitors. Threads execute in parallel, and communicate via shared objects that can be locked using synchronized access (a keyword in Java) to achieve mutual exclusion. However, with concurrent programming comes a new set of problems that can hamper the quality of the software. Deadlocks form such a problem category. In [16] a *deadlock* is defined as follows: "*Two or more threads block each other in a vicious cycle while trying to access synchronization locks needed to continue their activities*". That deadlocks pose a common problem is emphasized by the following statement in [16]: "*Among the most central and subtle liveness failures is deadlock. Without care, just about any design using synchronization on multiple cooperating objects can contain the possibility of deadlock*". Most of NASA's software that controls planetary rovers and space crafts is concurrent, and hence therefore poses a risk to mission success.

The difficulty in detecting deadlocks comes from the fact that concurrent programs typically are non-deterministic: several executions of the same program on the same input may yield different behaviors due to slight differences in the way threads are scheduled. This means in particular that generating the particular executions that expose a deadlock is difficult. Various technologies have been developed by the formal methods community to circumvent this problem, such as static analysis and, most recently, model checking. Static analysis, such as performed by tools like JLint [2] and ESC [7], analyze the source code without executing it. These techniques can be very efficient, but yield many false positives (false warnings) and additionally cannot well analyze programs where the object structure is very dynamic. Model checking has recently been applied to software (in contrast to only designs), for example in the Java PathFinder system (JPF) developed by NASA [12, 20], and in similar systems [9, 14, 6, 3, 19]. A model checker explores all possible execution paths of the program, and will therefore theoretically eventually expose a potential deadlock. This process is, however, quite resource demanding, in memory consumption as well in execution time, especially for large realistic programs consisting of thousands of lines of code. Using model checking for deadlock analysis has been discussed by J. Corbett [5]

---

\*VERIMAG, Centre Equation – 2, avenue de Vignate, F-38610 Gières, France. [Bensalem@imag.fr](mailto:Bensalem@imag.fr)

†Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA. [havelund@email.arc.nasa.gov](mailto:havelund@email.arc.nasa.gov).

Typically static analysis and model checking both try to be complete in the sense of avoiding false negatives (missed errors): all possibilities are examined. Furthermore, model checking tries to be general in exploring all kinds of errors. In the development of tools there is sometimes a conflict between generality (an important theoretical criterion) and efficiency. In order to make the techniques accepted in practice an important strategy can be to identify simple sub-classes of properties, whose analysis is tractable. Deadlocks is such a sub-class. We shall in particular investigate a technique based on trace analysis: a program is instrumented to log synchronization events when executed. The algorithm then examines the log file, building a lock graph, which reveals deadlock potentials by containing cycles. This technique has previously been implemented in the commercial tool Visual Threads [10] and scales very well since an arbitrary execution trace can reveal deadlocks even though such do not occur during the execution. The approach is essentially to turn a property (deadlock freedom) into a highly testable property (cycle freedom), that has higher probability of being detected if violated. The algorithm, however, can give false positives (as well as false negatives), putting a burden on the user to refute such. Our goal is to reduce the amount of false positives reported by the algorithm, and for that purpose we have modified it as reported in this paper. The modified algorithm has been implemented in Java to analyze Java programs, but the principles and theory presented are universal and apply in full to concurrent programs written in other languages.

The paper is organized as follows. Section 2 introduces preliminary notation used throughout the rest of the paper. Section 3 defines the notion of deadlock, outlines how deadlocks can be introduced in Java programs, and then discusses different ways of analyzing programs for deadlocks such as static analysis and model checking. Trace analysis is then suggested as a solution with a purpose, and the notion of *testable property* is defined. Section 4 presents the algorithm in three stages, starting with the basic algorithm as it is imagined implemented in [10] (the algorithm is only explained in few words in [10]), and then continuing with two modifications, each reducing false positives. Section 5 shortly describes the implementation of the algorithm in the Java PathExplorer tool and presents the results of a couple of case studies. Finally, Section 6 contains conclusions.

## 2 Notations and Preliminaries

A labelled transition system is given by  $(Q, E, R)$ , where  $Q$  is the set of states,  $E$  the set of labels and  $R \subseteq Q \times E \times Q$  is the transition relation. A directed graph is a pair  $G = (S, R)$  of sets satisfying  $R \subseteq S \times S$ . The set  $R$  is called the edge set of  $G$ , and its elements are called edges. A path  $p$  is a non-empty graph  $G = (S, R)$  of the form  $S = \{x_1, x_2, \dots, x_k\}$  and  $R = \{(x_1, x_2), (x_2, x_3), \dots, (x_{k-1}, x_k)\}$ , where the  $x_i$  are all distinct, except that  $x_k$  may be equal to  $x_1$ , in which case the path is a cycle. The nodes  $x_0$  and  $x_k$  are linked by  $p$ ; we often refer to a path by the natural sequence of its nodes, writing, say,  $p = x_1, x_2, \dots, x_k$  and calling  $p$  a path from  $x_1$  to  $x_k$ . In case where the edges are labelled with elements in  $L$ ,  $G$  is triplet  $(S, L, R)$  and called a labelled graph with  $R \subseteq S \times L \times S$ . A labelled path, respectively cycle, is a labelled graph with the obvious meaning. Given a sequence  $\sigma = x_1, x_2, \dots, x_n$ , we refer to an element at the position  $i$  in  $\sigma$  by  $\sigma[i]$  and the length of  $\sigma$  by  $|\sigma|$ . We let  $\langle \rangle$  denote the empty sequence, and the concatenation of two sequences  $\sigma_1$  and  $\sigma_2$  is denoted by  $\sigma_1 \frown \sigma_2$ . We denote by  $\sigma^i$  the prefix  $x_1, \dots, x_i$ . Let  $M : [A \xrightarrow{m} B]$  be a finite domain mapping from elements in  $A$  to elements in  $B$  (the  $\xrightarrow{m}$  operator generates the set of finite domain mappings from  $A$  to  $B$ , hence partial functions on  $A$ ). We let  $M \dagger [a \mapsto b]$  denote the mapping  $M$  overwritten with  $a$  mapping to  $b$ . That is, the  $\dagger$  operator represents map overwriting, and  $[a \mapsto b]$  represents a map that maps  $a$  to  $b$ . Looking up the value mapped to by  $a$  in  $M$  is denoted by  $M[a]$ . We denote the empty mapping by  $\square$ .

## 3 Deadlock Detection in Multi-threaded Java Programs

Deadlock is one of the most serious problems in multitasking concurrent programming systems. As early as in the 60's the deadlock problem was recognized and analyzed, Dijkstra [8] described it as the problem of the *deadly embrace*. Two types of deadlocks have been discussed in the literature [18] [15]: *resource deadlocks* and *communication deadlocks*. In resource deadlocks, a process which requests resources must wait until it acquires all the requested resources before it can proceed with its computation. A set of processes is resource deadlocked if each process in the set requests a resource held by another process in the set. In communication deadlock, messages are the resources for which processes wait. Reception of a message takes a process out of wait. A set of processes is communication deadlocked if each process in the set is waiting for a message from another process in the set and no process in the set ever sends a message. In this paper we focus only on *resource deadlocks*. Formally the concept of deadlock can be defined as follows.

**Definition 1 (Deadlock)** : A deadlock can occur between  $n$  threads  $t_1, \dots, t_n$  if they access  $n$  shared locks  $L = \{l_1, \dots, l_n\}$  and there is a state of the execution, and an enumeration  $E$  of  $L$ , such that each  $t_i$  holds  $E(i)$  in that state and  $t_i$  next wants to take  $E(j)$  for some  $j \neq i$ .

Java [1] is a general purpose object oriented programming language with built in features for multi-threaded programming. Threads can communicate via shared objects by for example calling methods on those objects. In order to avoid data races in these situations (where several threads access a shared object simultaneously), objects can be locked using the `synchronized` statement, or by declaring methods on the shared objects `synchronized`, which is equivalent. For example, a thread  $t$  can obtain a lock on an object  $A$  and then execute a statement  $S$  while having that lock as follows: `synchronized(A){S}`. During the execution of  $S$ , no other thread can obtain a lock on  $A$ . However  $t$  is allowed to take the same lock recursively. The lock is released when the scope of the `synchronized` statement is left; that is, when execution passes the `}` bracket. Java also provides the `wait` and `notify` primitives in support for user controlled interleaving between threads. While the `synchronized` primitive is the main source for resource deadlocks in Java, the `wait` and `notify` primitives are the main source for communication deadlocks. Since this paper focuses on resource deadlocks, we shall in the following focus on Java's capability of creating and executing threads and on the `synchronized` statement.

### 3.1 Detecting Deadlocks By Analyzing Code

A multi-threaded Java program can naturally be analyzed by simply executing it, or an instrumented version of it, on an existing Java Virtual Machine. This is the solution that we shall eventually explore. However, in this section some alternative solutions will be examined, namely static analysis and model checking. Each tool was applied to the classical dining philosopher<sup>1</sup> example in Java, but none of the tools performed convincingly as shall be explained.

JLint [2] is a tool, that examines a Java program for a limited set of errors. The errors it can detect can be classified into sequential errors, such as null pointer references and array-out-of-bound errors; and concurrency errors, such as data races and deadlocks. When we applied JLint to the philosopher program, no warnings were emitted, in particular the deadlock potential was not detected. The main reason for this is the use of an array to store the forks and the use of the modulo-operator to create the cyclic ring of philosophers and forks. The program is simply "too dynamic" in its creation of locks for JLint to detect the problem.

Java PathFinder (JPF) [20] can analyze a Java program dynamically, by executing it (the class files) on a specialized Java Virtual Machine. JPF, however, not only explores a single execution path, but all execution paths, thereby exploring all possible interleavings of threads in the program. If a resource deadlock is possible, it will then eventually be reached. In order to minimize the search, JPF stores all reached states, and avoids the search of a subtree of a state if that subtree has already been explored before (the state is stored). JPF also uses various other techniques to minimize the search, such as heuristics for prioritizing execution paths. We used a particular heuristics called `most-blocked`, which should be suited for this problem. It causes JPF to maximize the number of threads that are blocked. For 21 philosophers<sup>2</sup>, JPF went out of memory after 4 minutes and 40 seconds, using 1.46 GB.

We finally tried to verify a version of the program that did not have a deadlock. This forces JPF to explore the entire state space, which of course reduces the amount of philosophers that can be analyzed. The modified version of the program contains a gate lock, say a shared salt shaker, which is taken as the first thing by all philosophers, before they take their forks, hence preventing the cyclic deadlocks. With 3 philosophers JPF verified the program correct (deadlock free) in 3 minutes, using 256MB. However, with 4 philosophers JPF went out of memory after 26 minutes, using 1,46GB. The example considered is of course not realistic, but illustrates the point: neither model checking, nor static analysis handles this example convincingly.

### 3.2 Detecting Deadlocks By Analyzing Traces

An alternative to the above mentioned code analysis techniques is to execute an instrumented version of the program, thereby obtaining an execution trace, and then regard this trace as a dynamic abstract model

<sup>1</sup>The experiments were performed on a 2.2 GHz DELL desktop with 2GB available memory, of which 1.5 GB were allocated for the experiment.

<sup>2</sup>For 15 (resp. 20) philosophers JPF found the deadlock in 32.4 seconds (resp. 2 minutes and 51 seconds) using 343MB (resp. 1,45 GB) of memory.

of the program that can be analyzed for deadlock symptoms. The assumption is that the program has not deadlocked, and hence the trace does not explicitly represent a deadlock situation. The goal is to determine whether one from the trace can deduce the existence of another execution (trace) that deadlocks. In particular, as will be explained in the following, one can apply model checking or specialized analysis (as in static analysis) to the dynamic model. The advantage of the dynamic model approach is that a dynamic model contains precise information (although only for one trace), whereas a static model as used in JLint typically only contains partial information (although for all traces).

When analyzing a program for deadlock potentials, we are interested in observing all lock acquisitions and releases. The program can be instrumented to emit such events whenever locks are taken and released. Specifically, we are interested in two types of events:  $l(t, o)$  and  $u(t, o)$ , which means respectively that thread  $t$  locks object  $o$ ; and thread  $t$  unlocks object  $o$ . A lock trace  $\sigma = e_1, e_2, \dots, e_n$  is a finite sequence of lock and unlock events. Let  $E_\sigma$  denote the set of events occurring in  $\sigma$ . Let  $T_\sigma$  denote the set of threads occurring in  $E_\sigma$ , and let  $L_\sigma$  denote the set of locks occurring in  $E_\sigma$ . In this paper we assume the existence of an execution trace  $\sigma$  obtained by running an instrumented program. We assume for convenience that the trace is *reentrant free* in the sense that an already acquired lock is never re-acquired by the same thread (or any other thread of course) before being released. Note that Java supports reentrant locks by allowing a lock to be re-taken by a thread that already has the lock. However, the instrumentation can generate reentrant free traces if it is recorded how many times a lock has been acquired by each thread.

In the following, two approaches to analyzing traces for deadlock symptoms will be outlined: model checking and specialized cycle detection algorithms. We shall conclude that specialized algorithms are to be preferred.

**Model Checking Traces** The idea is to apply model checking to the execution trace in order to examine all possible interleavings of the trace, and determine whether one of them reaches a deadlock state. This can be done as follows. First we project the trace on each thread in  $T_\sigma$ . This results in a trace for each thread, which contains exactly those events the thread contributed to the trace. Each such trace can be regarded as an abstract sequential program, denoting a corresponding transition system. The parallel composition of these transition systems can then be formed, and examined for deadlock states. This can be formalized as follows. First we define the projection of the trace  $\sigma$  on each thread in  $T_\sigma$ , resulting in a transition system.

**Definition 2 (Projected trace transition systems)** *Given an execution trace  $\sigma = e_1, e_2, \dots, e_n$  with  $T_\sigma = \{t_1, \dots, t_m\}$ . Let  $\sigma_{\downarrow t_i}$  be the projection of  $\sigma$  on  $t_i$  for  $i \in \{1, \dots, m\}$ , meaning the trace obtained by eliminating all events not performed by  $t_i$ . We associate a projected labelled transition system  $S_i = (Q_i, E_i, \longrightarrow_i)$  for each  $\sigma_{\downarrow t_i}$  such that :*

- $Q_i = \{1, 2, \dots, k_i\}$ , where  $k_i = |\sigma_{\downarrow t_i}| + 1$ ,
- $E_i$  is  $E_{\sigma_{\downarrow t_i}}$ , and
- $\longrightarrow_i \subseteq Q_i \times E_i \times Q_i$  is defined as  $\{(i, \sigma[i], i + 1) \mid i \in \{1, \dots, |\sigma_{\downarrow t_i}|\}\}$

The states of the transition system for a projected trace are the positions in the projected trace and the events are the events of the trace. The transition relation relates neighbor positions in the trace corresponding to a sequential execution semantics. The product of the obtained transition systems represents the parallel composition of these, and hence represents all possible interleavings of the lock and unlock events from different threads, respecting that locks can only be held by one thread at a time. The composed transition system is defined as follows.

**Definition 3 (Composed trace transition system)** *Given the projected transition systems  $S_i = (Q_i, E_i, \longrightarrow_i)$ ,  $i = 1, \dots, m$ , associated to  $\sigma_{\downarrow t_i}$ . We define the composition of the transition systems  $S_i$ , denoted by  $\parallel_{i=1}^m S_i$ , by  $(Q, E, \longrightarrow)$  where :*

- $Q = (Q_1 \times Q_2 \times \dots \times Q_m \times 2^{\lambda(E)})$ ,  $\lambda(E)$  is the set of the resources that occur in the events of  $E$ , and
- $E = E_1 \cup E_2 \cup \dots \cup E_m$ , and

- $\longrightarrow \subseteq Q \times E \times Q$  is defined by :

$$\frac{s_i \xrightarrow{l(t_i, o)} s'_i \wedge o \notin \mathcal{L}}{(s_1, \dots, s_i, \dots, s_m, \mathcal{L}) \xrightarrow{l(t_i, o)} (s_1, \dots, s'_i, \dots, s_m, \mathcal{L} \cup \{o\})} \quad (1)$$

$$\frac{s_i \xrightarrow{u(t_i, o)} s'_i}{(s_1, \dots, s_i, \dots, s_m, \mathcal{L}) \xrightarrow{u(t_i, o)} (s_1, \dots, s'_i, \dots, s_m, \mathcal{L} \setminus \{o\})} \quad (2)$$

A state of the composed transition system includes a set of locks that have been acquired so far. The transition relation defines the interleaved execution of the individual transition systems, updating this set when locks are taken and released. The effect of the set is to prevent a lock to be taken by more than one thread at a time. Hence, a thread cannot proceed if it needs to acquire a lock that is in the set.

An execution trace  $\sigma = e_1, e_2, \dots, e_n$  of  $\parallel_{i=1}^m S_i$  is a sequence of events such that there exist states  $s_1, \dots, s_n$  in  $Q$ , such that  $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} \dots \xrightarrow{e_n} s_n$ , where  $s_0 = (1, 1, \dots, 1, \{\})$  is the initial state, and  $s_n$  (the last state) can progress no further: there does not exist an event  $e$  and a state  $s_{n+1}$  such that  $s_n \xrightarrow{e} s_{n+1}$ . We let  $\Sigma$  denote the set of all execution traces of  $\parallel_{i=1}^m S_i$ . We say that a trace in  $\Sigma$  is *deadlocked* if the last state  $s_n$  is different from the final state where all threads have reached their final state:  $s_n \neq (k_1, k_2, \dots, k_m, \{\})$ . For such a deadlocked trace  $\sigma$  we further say for some thread  $t$  and lock  $o$  that:

- $t$  holds  $o$  in  $\sigma$  if there exists a position  $i$  such that  $\sigma[i] = l(t, o)$ , and there does not exist a position  $j > i$  such that  $\sigma[j] = u(t, o)$ .
- $t$  wants  $o$  in  $\sigma$  if the last state  $s_n = (\dots, s_i, \dots, \mathcal{L})$  and  $s_i \xrightarrow{l(t, o)} s'_i$ . Note that in this case  $o \in \mathcal{L}$ .

We say that the trace  $\sigma$  is deadlock free if the interleaved parallel execution of the projections is deadlock free in the sense of Definition 1. The following lemma states that this can be determined by model checking the composed transition system.

**Lemma 4 (Trace Model Checking for Deadlock Detection)** *Let  $\parallel_{i=1}^m S_i = (Q, E, \longrightarrow)$  be the composition of the transition systems  $S_i = (Q_i, E_i, \longrightarrow_i)$ ,  $i = 1, \dots, m$ , obtained from projecting the trace  $\sigma$  on the  $m$  threads in  $T_\sigma$ . Let  $\Sigma$  be the set of all execution traces of  $\parallel_{i=1}^m S_i$ . The trace  $\sigma$  is deadlock free if and only if there are no deadlocked traces in  $\Sigma$ .*

As an experiment applying this approach, we handcrafted a Java program corresponding to the parallel composition of the individual traces obtained by running the dining philosopher program. For 48<sup>3</sup> philosophers, JPF went out of memory after 6 minutes and 15 seconds, using 1.46 GB. Although these numbers are quite impressive for a model checker, the results are a lot worse in the case of a deadlock free program, where the model checker has to explore all the states of the program in order to give a verdict. For 3 philosophers JPF verified the deadlock free program (introducing a gate lock) correct in 38.6 seconds using 116.53 MB of memory. For 4 philosophers, however, JPF went out of memory after 35 minutes and 26 seconds, using 1.46 GB. Model checking the trace(s) amounts to become complexity wise the same as model checking an abstraction of the original program, where all statements except synchronization statements have been removed. That is, it compares to model checking a synchronization skeleton [3] or an abstraction [6] of the program. With more than 3 threads, we have seen that this problem can become intractable in practice in the case there are no deadlocks (although the approach seems to have some advantages). The next section explores an alternative.

**Turning Deadlock Freedom to a Testable Property** The alternative approach pursued in this paper consists of building (in linear time) a specialized lock graph from the trace, which is then analyzed for cycles. A cyclic dependency between locks suggests that there exists an execution trace of the program that may deadlock. This technique has been implemented in the Visual Threads tool [10]. The approach may yield

<sup>3</sup>For 25 (resp. 47) philosophers JPF found the deadlock in 14.4 seconds (resp. 5 minutes and 6 seconds) using 105.29 MB (resp. 1,42 GB) of memory.

false positives since such a deadlocking execution may not exist due to program logic not visible in the trace – as well as false negatives, since only one trace is examined. The approach is a particular instance of a general approach, where a property  $\varphi$  (in our case: deadlock freedom) is reformulated as a testable property  $\psi$  (in our case: cycle freedom), which with high probability  $n$  will fail on *any* random execution trace if and only if the program does not satisfy the original property  $\varphi$  for *some* trace. In the ideal case, the probability  $n$  is 1.

This ideal case can be formalized as follows. Assume a programming language  $L_p$  being a set of well-formed programs. Let each program  $P \in L_p$  denote a set of traces, namely the possible execution traces of  $P$ . We shall not assume any particular structure of the traces. We shall let  $\sigma \in P$  denote the fact that  $\sigma$  is a trace of  $P$ . Assume further a specification language  $L_s$  being a set of well-formed properties. Assume a satisfaction relation between traces and properties, where  $\sigma \models \varphi$  denotes the fact that the trace  $\sigma$  satisfies the property  $\varphi$ . We can lift this relation to a relation between programs and properties as follows:  $P \models \varphi$  iff.  $\forall \sigma \in P . \sigma \models \varphi$ . By  $P \not\models \varphi$  for some program  $P$  and some property  $\varphi$  we mean  $\neg(P \models \varphi)$ .

**Definition 5 (Testable property)** *We say a property  $\psi$  is a testable property of a program  $P$  w.r.t. a property  $\varphi$  if and only if:*

1. *if there exists a trace  $\sigma \in P$  such that  $\sigma \models \psi$ , then  $P \models \varphi$*
2. *if there exists a trace  $\sigma \in P$  such that  $\sigma \not\models \psi$ , then  $P \not\models \varphi$*

In particular 1 is equivalent to:  $P \not\models \varphi$  implies  $\forall \sigma \in P . \sigma \not\models \psi$ , which states the desirable property that if the program  $P$  does not satisfy the property  $\varphi$  (that is, there exists a trace which does not satisfy  $\varphi$ ) then no matter what execution trace we choose, this will be detected by verifying the property  $\psi$ . The notion of testable property is an ideal. In practice we can not rely on this idealized view. Even though the idealized testable property cannot be achieved, a property can be practically testable, meaning that the probability  $n$  has an acceptable size. In the following we shall present practically testable properties for deadlock-freedom based on the basic algorithm for testing cycle freedom in lock graphs. We shall extend this algorithm to avoid false positives of three different kinds, hence improving the precision of the algorithm.

## 4 Trace Algorithm Based on Testable Properties

In essence, the detection algorithm consists of finding cycles in a *lock graph*. In the context of multi-threaded programs, the basic algorithm sketched in [10] works as follows. The multi-threaded program under observation is executed, while *lock* and *unlock* events are observed. A graph of locks is built, with edges between locks symbolizing locking orders. Any cycle in the graph signifies a potential for a deadlock. In this section, we present first the basic algorithm and then we present two conservative extensions of this algorithm that reduce the amount of false positives. We start with a through-going example.

### 4.1 Basic Example

We shall with an example illustrate the three categories of false positives. The first category, *single threaded cycles*, refers to cycles that are created by one single thread. *Guarded cycles* refer to cycles that are guarded by a gate lock "taken higher" up by all involved threads. Finally, *thread segmented cycles* refer to cycles between thread segments that cannot possibly execute concurrently. The program in Figure 1 illustrates these three situations, and a true positive. The real deadlock potential exists between threads  $T_2$  and  $T_3$ , corresponding to a cycle on  $L_1$  and  $L_2$ . The single threaded cycle within  $T_1$  clearly does not represent a deadlock. The guarded cycle between  $T_1$  and  $T_2$  does not represent a deadlock since both threads must acquire the gate lock  $G$  first. Finally, the thread segmented cycle between  $T_1$  and  $T_3$  does not represent a deadlock since  $T_3$  will execute before  $T_1$  executes its last synchronization segment.

For illustration purposes we shall assume a non-deadlocking execution trace  $\sigma$  for this program. It doesn't matter which one since all non-deadlocking traces will reveal all four cycles in the program. We shall assume the following trace of line numbered events (the line number is the first argument), which first, after having started  $T_1$  and  $T_2$  from the *Main* thread, executes  $T_1$  until the join statement, then executes  $T_2$  to the end, then  $T_3$  to the end, and then continues with  $T_1$  after it has joined on  $T_3$ 's termination. The line numbers are given for illustration purposes, and are actually recorded in the implementation in order to provide the

*Main :*

```
01: new T1().start();
02: new T2().start();
```

<p><i>T</i><sub>1</sub> :</p> <pre>03: synchronized(G){ 04:   synchronized(L1){ 05:     synchronized(L2){} 06:   } 07: }; 08: t3 = new T3(); 09: t3.start(); 10: t3.join(); 11: synchronized(L2){ 12:   synchronized(L1){} 13: }</pre>	<p><i>T</i><sub>2</sub> :</p> <pre>14: synchronized(G){ 15:   synchronized(L2){ 16:     synchronized(L1){} 17:   } 18: }</pre>	<p><i>T</i><sub>3</sub> :</p> <pre>19: synchronized(L1){ 20:   synchronized(L2){} 21: }</pre>
--	--	---

Figure 1: Example containing four cycles, only one of which represents a deadlock potential

user with useful error messages. In addition to the lock and unlock events  $l(lno, t, o)$  and  $u(lno, t, o)$  for line numbers  $lno$ , threads  $t$  and locks  $o$ , the trace also contains events for thread start,  $s(lno, t_1, t_2)$  and thread join,  $j(lno, t_1, t_2)$ , meaning respectively that  $t_1$  starts or joins  $t_2$  in line number  $lno$ .

$$\sigma = s(1, Main, T_1), s(2, Main, T_2), l(3, T_1, G), l(4, T_1, L_1), l(5, T_1, L_2), u(5, T_1, L_2), u(6, T_1, L_1), u(7, T_1, G), \\ s(9, T_1, T_3), l(14, T_2, G), l(15, T_2, L_2), l(16, T_2, L_1), u(16, T_2, L_1), u(17, T_2, L_2), u(18, T_2, G), l(19, T_3, L_1), \\ l(20, T_3, L_2), u(20, T_3, L_2), u(21, T_3, L_1), j(10, T_1, T_3), l(11, T_1, L_2), l(12, T_1, L_1), u(12, T_1, L_1), u(13, T_1, L_2)$$

In the remaining part of Section 4, we shall present three algorithms for detecting lock cycles in traces, being increasingly precise in eliminating false positives. First we shall present the basic algorithm that yields all four cycles as warnings. The final algorithm yields only the true positive for this example, and no false positives.

## 4.2 Basic Cycle Detection Algorithm

We shall initially restrict ourselves to traces including only lock and unlock events (no start or join events). In order to define the lock graph, we introduce a notion that we call a *lock context* of a trace  $\sigma$  in position  $i$ , denoted by  $\mathcal{C}_L(\sigma, i)$ . It's a mapping from each thread to the set of locks owned by that thread at that position. Formally, for a thread  $t \in T_\sigma$  we have the following :

$$\mathcal{C}_L(\sigma, i)(t) = \{o \mid \exists j : j \leq i \wedge \sigma[j] = l(t, o) \wedge \neg \exists k : j < k \leq i \wedge \sigma[k] = u(t, o)\}$$

Bellow we give a definition that allows to build the lock graph  $G_L$  with respect to an execution trace  $\sigma$ . An edge in  $G_L$  between two locks  $l_1$  and  $l_2$  means that there exists a thread  $t$  which owns the object  $l_1$  while taking the object  $l_2$ .

**Definition 6 (Lock graph)** *Given an execution trace  $\sigma = e_1, e_2, \dots, e_n$ . We say that the lock graph of  $\sigma$  is the minimal directed graph  $G_L = (L, R)$  such that:*

- $L$  is the set of locks  $L_\sigma$ ,
- $R \subseteq L \times L$  is defined by  $(l_1, l_2) \in R$  if there exists a thread  $t \in T_\sigma$  and a position  $i \geq 2$  in  $\sigma$  such that :

$$\sigma[i] = l(t, l_2) \text{ and } l_1 \in \mathcal{C}_L(\sigma, i-1)(t)$$

The definition 6 above is declarative. In Figure 2 we give an algorithm for constructing the lock graph from a lock trace. In this algorithm, we also use the context  $\mathcal{C}_L$  which is exactly the same as in the definition 6. The only difference is that we don't need to use explicitly the two parameters  $\sigma$  and  $i$ . The set of cycles (Section 2) in the graph  $G_L$ , denoted by  $cycles(G_L)$ , represents the potential deadlock situations in the program. The lock graph for the example in Figure 1 is also shown in Figure 2.

**Input:** An execution trace  $\sigma$   
 $G_L$  is a graph;  
 $\mathcal{C}_L : [T_\sigma \rightarrow 2^{L_\sigma}]$  is a lock context;  
**for** ( $i = 1 \dots |\sigma|$ ) **do**  
    **case**  $\sigma[i]$  **of**  
         $l(t, o) \rightarrow$   
             $G_L := G_L \cup \{(o', o) \mid o' \in \mathcal{C}_L(t)\};$   
             $\mathcal{C}_L := \mathcal{C}_L \uparrow [t \mapsto \mathcal{C}_L(t) \cup \{o\}];$   
         $u(t, o) \rightarrow$   
             $\mathcal{C}_L := \mathcal{C}_L \uparrow [t \mapsto \mathcal{C}_L(t) \setminus \{o\}]$   
    **end;**  
**for each**  $c$  in  $\text{cycles}(G_L)$  **do**  
    **print** ("deadlock potential:",  $c$ );

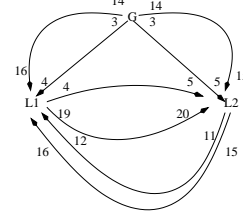


Figure 2: The basic algorithm and the lock graph

### 4.3 Eliminating Single Threaded Cycles and Guarded Cycles

In this section we present an algorithm that removes false positives stemming from *single threaded cycles* and *guarded cycles*. In [11] a solution was suggested, based on building synchronization trees. However, this solution could only detect deadlocks between pairs of threads. The algorithm to be presented here is not limited in this sense. The solution is to extend the lock graph by labelling each edge between locks with information about which thread causes the addition of the edge and what gate locks were held by that thread when the target lock was taken. The definition of *valid cycles* will then include this information to filter out false positives. First, we define the extended lock graph.

**Definition 7 (Guarded lock graph)** *Given an execution trace  $\sigma = e_1, e_2, \dots, e_n$ . We say that the guarded lock graph of  $\sigma$  is the minimal directed labelled graph  $G_L = (L, W, R)$  such that:*

- $L$  is the set of locks  $L_\sigma$
- $W = T_\sigma \times 2^L$  is the set of labels, each containing a thread id and a lock set,
- $R \subseteq L \times W \times L$  is defined by  $(l_1, (t, g), l_2) \in R$  if there exists a thread  $t \in T_\sigma$  and a position  $i \geq 2$  in  $\sigma$  such that:

$$\sigma[i] = l(t, l_2) \text{ and } l_1 \in \mathcal{C}(\sigma, i-1)(t) \text{ and } g = \mathcal{C}(\sigma, i-1)(t)$$

Each edge  $(l_1, (t, g), l_2)$  in  $R$  is labelled with the thread  $t$  that took the locks  $l_1$  and  $l_2$ , and a lock set  $g$ , indicating what locks  $t$  owned when taking  $l_2$ . In order for a cycle to be valid, and hence regarded as a true positive, the threads and guard sets occurring in labels of the cycle must be valid in the following sense:

**Definition 8 (Valid threads and guards)** *Let  $G_L$  be a guarded lock graph of some execution trace and  $c = (L, W, R)$  a cycle in  $\text{cycles}(G_L)$ , we say that:*

- threads of  $c$  are valid if for all labels  $e, e' \in W$   $e \neq e'$  implies  $\text{thread}(e) \neq \text{thread}(e')$
- guards of  $c$  are valid if for all labels  $e, e' \in W$   $e \neq e'$  implies  $\text{guards}(e) \cap \text{guards}(e') = \emptyset$

where, for a label  $e \in W$ ,  $\text{thread}(e)$ , resp.  $\text{guards}(e)$ , gives the first, resp. second, component of  $e$ .

For a cycle to be valid, the threads involved must differ. This eliminates single threaded cycles. Furthermore, the lock sets on the edges in the cycle must not overlap. This eliminates cycles that are guarded by the same lock taken "higher up" by at least two of the threads involved in the cycle. Assume namely that such a gate lock exists, then it will belong to the lock sets of several edges in the cycle, and hence they will overlap at least on this lock. This corresponds to the fact that a deadlock cannot happen in this situation. Valid cycles are now defined as follows:



**Definition 9 (Guarded cycles)** Let  $\sigma$  be an execution trace and  $G_L$  its guarded lock graph. We say that a cycle  $c \in \text{cycles}(G_L)$  is a guarded cycle if the guards of  $c$  are valid and threads of  $c$  are also valid. We denote by  $\text{cycles}_g(G_L)$  the set of guarded cycles in  $\text{cycles}(G_L)$ .

We shall in this section not present an explicit algorithm for constructing this graph, since its concerns a relatively simple modification to the basic algorithm: the statement that updates the lock graph becomes:  $G_L := G_L \cup \{(\sigma', (t, \mathcal{C}(t)), \sigma) \mid \sigma' \in \mathcal{C}(t)\}$ , adding the labels  $(t, \mathcal{C}(t))$  to the edges. Furthermore, cycles to be reported should be drawn from:  $\text{cycles}_g(G_L)$ .

Let us illustrate the algorithm with an example. We consider again the execution trace  $\sigma$  presented in Subsection 4.1. The guarded graph for this trace is shown in Figure 3. The graph contains the same number of edges as the basic graph in Figure 2. However, now edges are labelled with a thread and a guard set. In particular, we notice that the gate lock  $G$  occurs in the guard set of edges (4, 5) and (15, 16). This prevents this guarded cycle from being included in the set of valid cycles since it is not guard valid: the guard sets overlap in  $G$ . Also the single threaded cycle (4, 5)  $\leftrightarrow$  (11, 12) is eliminated because it is not thread valid: the same thread  $T_1$  occurs on both edges.

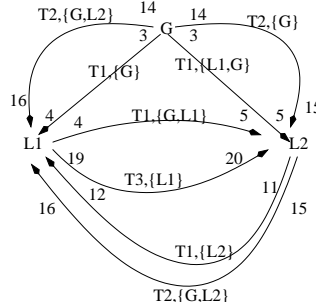


Figure 3: Guarded lock graph

The correctness of the guarded algorithm is stated in the following theorem, which states that any valid cycle reported by the algorithm for a trace  $\sigma$  corresponds to a deadlock situation in the composed transition system, and vice versa. We say that a (deadlocked) execution trace  $\sigma$  *reflects* a cycle  $c = (L, W, R)$  if for all  $(l_1, (t, G), l_2) \in R$ ,  $t$  holds  $l_1$  in  $\sigma$ , and  $t$  wants  $l_2$  in  $\sigma$  (see Section 3.2 page 5 for a definition of these terms).

**Theorem 10 (Correctness of guarded cycles)** Let  $\sigma$  be an execution trace,  $G_L$  its guarded lock graph, and  $\text{cycles}_g(G_L)$  the set of the guarded cycles. Let  $\Sigma$  be the set of all the execution traces of the system  $\|_{i=1}^m S_i$ , where the transition systems  $S_i$ ,  $i = 1, \dots, m$ , are obtained from projecting the trace  $\sigma$  on the  $m$  threads in  $T_\sigma$ . Then:

- for all cycles  $c \in \text{cycles}_g(G_L)$ , there exists an execution trace  $\sigma'$  in  $\Sigma$ , such that  $\sigma'$  is deadlocked and reflects  $c$  (no false positives with respect to  $\sigma$ ).
- for all traces  $\sigma'$  in  $\Sigma$ , if  $\sigma'$  is deadlocked, then there exists a cycle  $c \in \text{cycles}_g(G_L)$ , such that  $\sigma'$  reflects  $c$  (no false negatives with respect to  $\sigma$ ).

Note that when the above theorem states that there are no false positives or negatives, it is with respect to the execution trace  $\sigma$ . There may still be false positives and negatives with respect to the program, the execution of which resulted in the trace. The guarded deadlock algorithm may in rare cases miss deadlocks that the basic algorithm finds. As an example consider the program in Figure 1, and consider that the guard  $G$  in  $T_2$  is computed as the result of a conditional statement, in one run it may be  $G$ , while in another run it may be  $G'$ , different from  $G$ . In the latter case, the cycle between  $L_1$  and  $L_2$  in threads  $T_1$  and  $T_2$  is not guarded and there is a deadlock potential. The basic algorithm will detect this irrespective of whether  $G$  or  $G'$  is chosen, while the guarded algorithm will not in case  $G$  is chosen. Due to this observation one could report even guarded cycles, but marking them as possibly false positives.

#### 4.4 Eliminating Segmented Cycles

In the previous section we saw the specification of an algorithm that removes false positives stemming from single threaded cycles and guarded cycles. In this section we present an algorithm that furthermore removes false positives stemming from *segmented cycles*. We assume that traces now also contain start and join events. Recall the example in Figure 1 and that the basic algorithm reports a cycle between threads  $T_1$  (line 11-12) and  $T_3$  (line 19-20) on locks  $L_1$  and  $L_2$ . However, a deadlock is impossible since thread  $T_3$  is joined on by  $T_1$  in line 10. Hence, the two *code segments*: line 11-12 and line 19-20 can never run in parallel. The algorithm to be presented will prevent such cycles from being reported by formally introducing such a notion of *segments* that cannot execute in parallel. A new directed segmentation graph will record which segments execute before others. The lock graph is then extended with extra label information, that specifies what segments locks are acquired in, and the validity of a cycle now incorporates a check that the lock acquisitions are really occurring in parallel executing segments. The idea of using segmentation in runtime analysis was initially suggested in [10] to reduce the amount of false positives in data race analysis using the Eraser algorithm [17]. We use it in a similar manner here to reduce false positives in deadlock detection.

More specifically, the solution is during execution to associate segment identifiers (natural numbers, starting from 0) to segments of the code that are separated by statements that *start* or *join* other threads. For example, if a thread  $t_1$  currently is in segment  $s$  and starts another thread  $t_2$ , and the next free segment is  $n$ , then  $t_1$  will continue in segment  $n$  and  $t_2$  will start in segment  $n + 1$  (it could have been chosen differently, the main point being that new segments are allocated). From then on the next free segment will be  $n + 2$ . It is furthermore recorded in the segmentation graph that segment  $s$  executes before  $n$  as well as before  $n + 1$ . In a similar way, if a thread  $t_1$  currently is in segment  $s_1$  and joins another thread  $t_2$  that is in segment  $s_2$ , and the next free segment is  $n$ , then  $t_1$  will continue in segment  $n$ ,  $t_2$  will be terminated, and from then on the next free segment will be  $n + 1$ . It is recorded that  $s_1$  as well as  $s_2$  execute before  $n$ . Figure 5 illustrates the segmentation graph for the program example in Figure 1. Below we shall formalize these concepts, and finally suggest an algorithm.

In order to give a formal definition of the segmentation we need to define two functions. The first one,  $\mathcal{C}_S(\sigma)$ , *segmentation context* of the trace  $\sigma$ , gives for each position  $i$  of the execution trace  $\sigma$ , the current segment of each thread  $t$  at that position. Formally,  $\mathcal{C}_S(\sigma)$  is the mapping with type:  $[\mathcal{N} \mapsto [T_\sigma \mapsto \mathcal{N}]]$ , associated to trace  $\sigma$ , that maps each position into another mapping that maps each thread id to its current segment in that position. It is defined as follows. Let  $\mathcal{C}_S^{init} = [0 \mapsto [main \mapsto 0]]$ , mapping position 0 to the mapping that maps the main thread to segment 0. Then  $\mathcal{C}_S(\sigma)$  is defined by the use of the auxiliary function  $f_0 : \text{Trace} \times \text{Context} \times \text{Position} \times \text{Current\_Segment} \rightarrow \text{Context}$ :

$$\mathcal{C}_S(\sigma) = f_0(\sigma, \mathcal{C}_S^{init}, 1, 0) \quad (3)$$

$$f_0(e \frown \sigma, \mathcal{C}_S, i, n) = \begin{cases} f_0(\sigma, \mathcal{C}_S, i + 1, n) & \text{if } e \in \{l(t, o), u(t, o)\}, \\ f_0(\sigma, \mathcal{C}_S \uparrow [i \mapsto \mathcal{C}_S[i - 1] \uparrow \left[ \begin{smallmatrix} t_1 \mapsto n + 1 \\ t_2 \mapsto n + 2 \end{smallmatrix} \right], i + 1, n + 2) & \text{if } e = s(t_1, t_2), \\ f_0(\sigma, \mathcal{C}_S \uparrow [i \mapsto \mathcal{C}_S[i - 1] \uparrow [t_1 \mapsto n + 1], i + 1, n + 1) & \text{if } e = j(t_1, t_2). \end{cases} \quad (4)$$

$$f_0(<>, \mathcal{C}_S, i, n) = \mathcal{C}_S \quad (5)$$

The second function needed,  $\#_{alloc}$ , gives the number of segments allocated in position  $i$  of  $\sigma$ . This function is used to calculate what is the next segment to be assigned to a new execution block (the  $ns$  in the above example), and is dependent on the number of start events  $s(t_1, t_2)$  and join events  $j(t_1, t_2)$  that occur in the trace up and until position  $i$ , recalling that each start event causes two new segments to be allocated. Formally we define it as follows :  $\#_{alloc}(\sigma, i) = |\sigma^i \downarrow_s| * 2 + |\sigma^i \downarrow_j|$ .

We can now define the notion of a directed *segmentation graph*, which defines an ordering between segments. Informally, assume that in trace position  $i$  a thread  $t_1$ , being in segment  $s_1 = \mathcal{C}_S(\sigma)(i - 1)(t_1)$ , executes a start of a thread  $t_2$ . Then  $t_1$  continues in segment  $n = \#_{alloc}(\sigma, i - 1) + 1$  and  $t_2$  continues in segment  $n + 1$ . Consequently,  $(s_1, n)$  as well as  $(s_1, n + 1)$  belongs to the graph, meaning that  $s_1$  executes before  $n$  as well as before  $n + 1$ . Similarly, assume that a thread  $t_1$  in position  $i$ , being in segment  $s_1 = \mathcal{C}_S(\sigma)(i - 1)(t_1)$ , executes a join of a thread  $t_2$ , being in segment  $s_2 = \mathcal{C}_S(\sigma)(i - 1)(t_2)$ . Then  $t_1$  continues in segment  $n = \#_{alloc}(\sigma, i - 1) + 1$  while  $t_2$  terminates. Consequently  $(s_1, n)$  as well as  $(s_2, n)$  belongs to the graph, meaning that  $s_1$  as well as  $s_2$  executes before  $n$ . The formal definition of the segmentation graph is as follows.

**Definition 11 (Segmentation graph)** Given an execution trace  $\sigma = e_1, e_2, \dots, e_n$ . We say that a segmentation graph of  $\sigma$  is the directed graph  $G_S = (\mathcal{N}, R)$  where

- $\mathcal{N} = \{n \mid 0 \leq n \leq \#_{alloc}(\sigma, |\sigma|)\}$  is the set of segments
- $R \subseteq \mathcal{N} \times \mathcal{N}$  is the relation given by  $(s_1, s_2) \in R$  if there exists a position  $i \geq 1$  such that
 
$$\begin{aligned} \sigma[i] &= s(t_1, t_2) \wedge s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1) \wedge (s_2 = \#_{alloc}(\sigma, i-1) + 1 \vee s_2 = \#_{alloc}(\sigma, i-1) + 2) \\ \text{or} \\ \sigma[i] &= j(t_1, t_2) \wedge (s_1 = \mathcal{C}_S(\sigma)(i-1)(t_1) \vee s_1 = \mathcal{C}_S(\sigma)(i-1)(t_2)) \wedge s_2 = \#_{alloc}(\sigma, i-1) + 1 \end{aligned}$$

Given a segmentation graph, we can now define what it means for a segment to *happen before* another segment, reflecting how the segments are related in time during execution.

**Definition 12 (Happens-Before relation)** Let  $G_S = (\mathcal{N}, R)$  be a segmentation graph, and  $G_S^* = (\mathcal{N}, R^*)$  its transitive closure. Then given two segments  $s_1$  and  $s_2$ , we say that  $s_1$  happens before  $s_2$ , denoted by  $s_1 \triangleright s_2$ , if  $(s_1, s_2) \in R^*$ .

Note that for two given segments  $s_1$  and  $s_2$ , if neither  $s_1 \triangleright s_2$  nor  $s_2 \triangleright s_1$ , then we say that  $s_1$  happens in parallel with  $s_2$ . Before we can finally define what is a lock graph with segment information, we need to redefine the notion of lock context,  $\mathcal{C}_L(\sigma, i)$ , of a trace  $\sigma$  and a position  $i$ , that was defined on page 7. In the previous definition it was a mapping from each thread to the set of locks owned by that thread at that position. Now we add information about what segment each lock was taken in. Formally, for a thread  $t \in T_\sigma$  we have the following :

$$\begin{aligned} \mathcal{C}_L(\sigma, i)(t) = \\ \{(o, s) \mid \exists j : j \leq i \wedge \sigma[j] = l(t, o) \wedge \mathcal{C}_S(\sigma)(j)(t) = s \wedge \neg \exists k : j < k \leq i \wedge \sigma[k] = u(t, o)\} \end{aligned}$$

We can now give a definition of a lock graph  $G_L$  with respect to an execution trace  $\sigma$ , that contains segment information as well as gate lock information. An edge in  $G_L$  between two locks  $l_1$  and  $l_2$  means, as before, that there exists a thread  $t$  which owns an object  $l_1$  while taking the object  $l_2$ . The edge is as before labelled with  $t$  as well as the set of (gate) locks. In addition, the edge is now further labelled with the segments  $s_1$  and  $s_2$  in which the locks  $l_1$  and  $l_2$  were taken by  $t$ .

**Definition 13 (Segmented and guarded lock graph)** Given an execution trace  $\sigma = e_1, e_2, \dots, e_n$ . We say that the segmented and guarded lock graph of  $\sigma$  is the minimal directed graph  $G_L = (L_\sigma, W, R)$  such that:

- $W = \mathcal{N} \times (T_\sigma \times 2^{L_\sigma}) \times \mathcal{N}$  is the set of labels  $(s_1, (t, g), s_2)$ , each containing the segment  $s_1$  that the source lock was taken in, a thread id  $t$ , a lock set  $g$  (these two being the labels of the guarded lock graph in the previous section), and the segment  $s_2$  that the target lock was taken in,
- $R \subseteq L_\sigma \times W \times L_\sigma$  is defined by  $(l_1, (s_1, (t, g), s_2), l_2) \in R$  if there exists a thread  $t \in T_\sigma$  and a position  $i \geq 2$  in  $\sigma$  such that:

$$\begin{aligned} \sigma[i] &= l(t, l_2) \text{ and} \\ (l_1, s_1) &\in \mathcal{C}_L(\sigma)(i-1)(t) \text{ and} \\ g &= \{l' \mid (l', s) \in \mathcal{C}_L(\sigma)(i-1)(t)\} \text{ and} \\ s_2 &= \mathcal{C}_S(\sigma)(i-1)(t) \end{aligned}$$

Each edge  $(l_1, (s_1, (t, g), s_2), l_2)$  in  $R$  is labelled with the thread  $t$  that took the locks  $l_1$  and  $l_2$ , and a lock set  $g$ , indicating what locks  $t$  owned when taking  $l_2$ . Furthermore, the segments  $s_1$  and  $s_2$  indicate in which segments respectively  $l_1$  and  $l_2$  were taken.

In order for a cycle to be valid, and hence regarded as a true positive, the threads and guard sets occurring in labels of the cycle must be valid as before. In addition, the segments in which locks are taken must now allow for a deadlock to actually happen. Consider for example a cycle between two threads  $t_1$  and  $t_2$  on two locks  $l_1$  and  $l_2$ . Assume further that  $t_1$  takes  $l_1$  in segment  $x_1$  and then  $l_2$  in segment  $x_2$  while  $t_2$  takes them in opposite order, in segments  $y_1$  and  $y_2$  respectively. Then it must be possible for  $t_1$  and  $t_2$  to each

take their first lock in order for a deadlock to occur. In other words,  $x_2$  must not happen before  $y_1$  and  $y_2$  must not happen before  $x_1$ . This is expressed in the following definition, which repeats the definitions from Definition 8.

**Definition 14 (Valid threads, guards and segments)** *Let  $G_L$  be a segmented and guarded lock graph of some execution trace and  $c = (L, W, R)$  a cycle in  $\text{cycles}(G_L)$ , we say that:*

- *threads of  $c$  are valid if for all labels  $e, e' \in W$ ,  $e \neq e'$  implies  $\text{thread}(e) \neq \text{thread}(e')$*
- *guards of  $c$  are valid if for all labels  $e, e' \in W$ ,  $e \neq e'$  implies  $\text{guards}(e) \cap \text{guards}(e') = \emptyset$*
- *segments of  $c$  are valid if for all labels  $e, e' \in W$ ,  $e \neq e'$  implies  $\neg (\text{seg}_2(e_1) \triangleright \text{seg}_1(e_2))$*

where, for a label  $e = (s_1, (t, g), s_2) \in W$ ,  $\text{thread}(e) = t$ ,  $\text{guards}(e) = g$ ,  $\text{seg}_1(e) = s_1$  and  $\text{seg}_2(e) = s_2$ .

Valid cycles are now defined as follows.

**Definition 15 (Segmented and guarded cycles)** *Let  $\sigma$  be an execution trace and  $G_L$  its segmented and guarded lock graph. We say that a cycle  $c \in \text{cycles}(G_L)$  is a segmented and guarded cycle if the guards of  $c$  are valid, the threads of  $c$  are valid, and the segments of  $c$  are valid. We denote by  $\text{cycles}_s(G_L)$  the set of segmented and guarded cycles in  $\text{cycles}(G_L)$ .*

The definitions of segmentation graph (Definition 11) and segmented and guarded lock graph (Definition 13) above are declarative. Figure 4 presents an algorithm for constructing the segmentation graph and lock graph from an execution trace. The set of cycles in the graph  $G_L$ , denoted by  $\text{cycles}_s(G_L)$ , see Definition 15, represents the potential deadlock situations in the program. The segmentation graph ( $G_S$ ) and lock graph ( $G_L$ ) have the structure as outlined in Definition 11 and Definition 13 respectively. The lock context ( $C_L$ ) maps each thread to the set of locks owned by that thread at any point in time. Associated with each such lock is the segment in which it was acquired. The segment context ( $C_S$ ) maps each thread to the segment in which it is currently executing. The algorithm should after this explanation and the previously given abstract definitions be self explanatory.

Let us illustrate the algorithm with our example. We consider again the execution trace  $\sigma$  presented in Subsection 4.1. The segmentation graph for this trace is shown in Figure 5 and the segmented and guarded lock graph is shown in Figure 4. The segmentation graph is for illustrative purposes augmented with the statements that caused the graph to be updated. We see in particular that segment 6 of thread  $T_3$  executes before segment 7 of thread  $T_1$ , written as  $6 \triangleright 7$ . Segment 6 is the one in which  $T_3$  executes lines 19 and 20, while segment 7 is the one in which  $T_1$  executes lines 11 and 12. The lock graph contains the same number of edges as the guarded graph in Figure 3, and the same *(thread, guard set)* labels. However, now edges are additionally labelled with the segments in which locks are taken. This makes the cycle  $(19, 20) \leftrightarrow (11, 12)$  segment invalid since the target segment of the first edge (6) executes before the source segment of the second edge (7).

Concerning the correctness of the algorithm, a theorem similar to Theorem 10 can be formulated. However, the notion of composed transition system, as formulated in Definition 3, must be changed to incorporate *start* and *join* events. We shall not do that here, but just mention that two new rules must be added: one for start events  $s(t_1, t_2)$  that adds the initial state of thread  $t_2$  to the state, and one for join events  $j(t_1, t_2)$ , that is conditioned with the terminated status of  $t_2$ . We say that an execution trace  $\sigma$  *reflects* a cycle  $c = (L, W, R)$  if for all  $(l_1, (s_1, (t, g), s_2), l_2) \in R$ ,  $t$  holds  $l_1$  in  $\sigma$ , and  $t$  wants  $l_2$  in  $\sigma$  (see Section 3.2 page 5 for a definition of these terms). The correctness is now stated as follows (equivalent in formulation to Theorem 10, except for the use of  $\text{cycles}_s$  instead of  $\text{cycles}_g$ ).

**Theorem 16 (Correctness of segmented and guarded cycles)** *Let  $\sigma$  be an execution trace,  $G_L$  its segmented and guarded lock graph and  $\text{cycles}_s(G_L)$  the set of segmented and guarded cycles. Let  $\Sigma$  be the set of execution traces of the system  $\parallel_{i=1}^m S_i$ , where the transition systems  $S_i$ ,  $i = 1, \dots, m$ , are obtained from projecting the trace  $\sigma$  on the  $m$  threads in  $T_\sigma$ . Then:*

- *for all cycles  $c \in \text{cycles}_s(G_L)$ , there exists an execution trace  $\sigma'$  in  $\Sigma$ , such that  $\sigma'$  is deadlocked and reflects  $c$  (no false positives with respect to  $\sigma$ ).*
- *for all traces  $\sigma'$  in  $\Sigma$ , if  $\sigma'$  is deadlocked, then there exists a cycle  $c \in \text{cycles}_s(G_L)$ , such that  $\sigma'$  reflects  $c$  (no false negatives with respect to  $\sigma$ ).*

**Input:** An execution trace  $\sigma$   
 $G_L$  is a lock graph;  
 $G_S$  is a segmentation graph;  
 $C_L : [T_\sigma \rightarrow 2^{L_\sigma \times \mathbf{nat}}]$  is a lock context;  
 $C_S : [T_\sigma \rightarrow \mathbf{nat}]$  is a segment context;  
 $n : \mathbf{nat} = 1$  next available segment;  
**for** ( $i = 1 \dots |\sigma|$ ) **do**  
  **case**  $\sigma[i]$  **of**  
     $l(t, o) \rightarrow$   
       $G_L := G_L \cup \{(o', (s_1, (t, g), s_2), o) \mid$   
        $(o', s_1) \in C_L(t) \wedge$   
        $g = \{o'' \mid (o'', s) \in C_L(t)\} \wedge$   
        $s_2 = C_S(t)\};$   
       $C_L := C_L \uparrow [t \mapsto C_L(t) \cup \{(o, C_S(t))\}];$   
     $u(t, o) \rightarrow$   
       $C_L := C_L \uparrow [t \mapsto C_L(t) \setminus \{(o, *)\}];$   
     $s(t_1, t_2) \rightarrow$   
       $G_S := G_S \cup \{(C_S(t_1), n), (C_S(t_1), n + 1)\};$   
       $C_S := C_S \uparrow [t_1 \mapsto n, t_2 \mapsto n + 1];$   
       $n := n + 2;$   
     $j(t_1, t_2) \rightarrow$   
       $G_S := G_S \cup \{(C_S(t_1), n), (C_S(t_2), n)\};$   
       $C_S := C_S \uparrow [t_1 \mapsto n];$   
       $n := n + 1;$   
  **end;**  
**for each**  $c$  in  $\text{cycles}_s(G_L)$  **do**  
  **print** ("deadlock potential:",  $c$ );

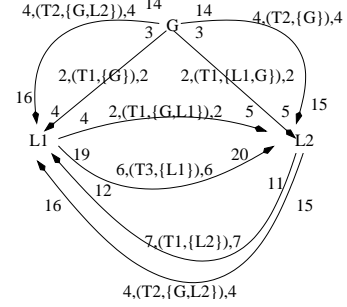


Figure 4: The final algorithm and the segmented lock graph

## 5 Implementation and Experimentation

The algorithm presented in Section 4.4 has been implemented in the Java PathExplorer tool [13], in short referred to as JPaX. JPaX analyzes Java programs for deadlocks, using the presented algorithm, and for data races, using a homegrown adaption of the Eraser algorithm [17] to work for Java. In the following we shall primarily focus on the deadlock analysis. JPaX itself is written in Java, and consists of two main modules, an *instrumentation module* and an *observer module*. The instrumentation module automatically instruments the bytecode class files of a compiled program by adding new instructions that when executed generate the execution trace consisting of the events needed for the analysis. In our case lock events  $l(t, o)$  and unlock events  $u(t, o)$ , together with start events  $s(t_1, t_2)$  and join events  $j(t_1, t_2)$  are generated. The generated events are either sent to a socket or written to a file (in both cases in plain text format), depending on whether the analysis should be on-the-fly, during the execution of the analyzed program, or whether it is acceptable that it is performed after the analyzed program has terminated. The file solution has been the one most frequently used in our case studies. The observer module consequently reads the event stream and dispatches this to a set of observer rules, each rule performing a particular analysis that has been requested, such as deadlock analysis and data race analysis. This modular rule based design allows a user to easily define new runtime verification procedures without interfering with legacy code.

The Java bytecode instrumentation is performed using the Jtrek Java bytecode engineering tool [4]. Jtrek makes it possible to easily read Java class files (bytecode files), and traverse them as abstract syntax trees while examining their contents, and insert new code. The inserted code can access the contents of various runtime data structures, such as for example the call-time stack, and will, when eventually executed, emit events carrying this extracted information to the observer. As already mentioned, this form of analysis is not complete and hence may yield false negatives by missing to report synchronization problems. A synchronization problem can most obviously be missed if one or more of the synchronization statements involved in the problem do not get executed. To avoid being entirely in the dark in these situations, we added a coverage module to the system that records what lock-related instructions are instrumented and

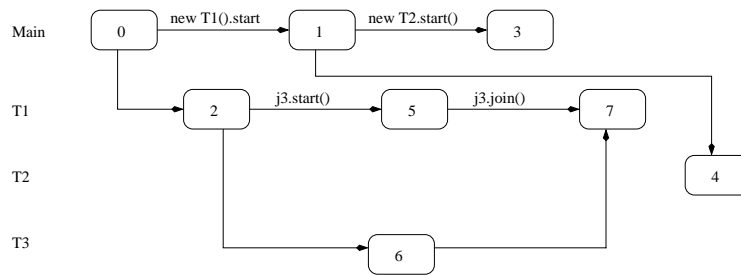


Figure 5: Segmentation graph

which of these that are actually executed. The difference is printed as part of the error report for the user to react on, for example by generating better test cases.

JPaX has been applied to two case studies at NASA Ames: a planetary rover controller (named K9), and a space craft altitude control system (ACS), both being translated to Java from C++ and C respectively as part of an attempt to evaluate Java for mission software. Two resource deadlocks and two data races were seeded in the rover code by an independent team. JPaX found them all. In addition, an early version of the deadlock algorithm found a deadlock in the original C++ version of K9 that was unexpected by the programmer. This experiment was performed by creating a C++ specific instrumentation module, whereas the observer module could be used unmodified. In ACS, JPaX found 2 unexpected data races and 2 seeded data races. We also applied the JPaX deadlock analysis algorithm to the classical dining philosopher example. For the deadlocking version, for 100 philosophers, JPaX found the deadlock in 8 seconds, including instrumentation. For 300 JPaX found the deadlock in 22 seconds. For the deadlock free version, for 4 philosophers JPaX concluded correctness in 7 seconds, for 100 in 30 seconds, and for 300 in 2 minutes, out of which 40 seconds were due to a slowdown in the running program due to instrumentation.

## 6 Conclusions

An algorithm has been presented for detecting deadlock potentials in concurrent programs by analyzing execution traces. The algorithm extends an existing algorithm by reducing the amount of false positives reported, and has been implemented in the Java PathExplorer tool that in addition to deadlocks also analyzes for data races and for consistency with user provided temporal properties. Although JPaX analyzes Java programs, it can be applied to applications written in other languages by modifying the instrumentation module. The advantage of trace analysis is that it scales extremely well, in contrast to more formal methods, and in addition can detect errors that for example static analysis cannot properly detect. In future work, we expect to approach the problem of false negatives (missed errors) by developing a framework for symbolically inferring what test cases are needed to exercise all synchronization statements in a program. At an extreme, static analysis of deadlocks can be combined with dynamic analysis. Current work attempts to extend the capabilities of JPaX with new algorithms for detecting other kinds of concurrency errors, such as other forms of data races and communication deadlocks. An additional important issue that we will address is the performance impact on the instrumented program.

## References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
- [2] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *13th Australian Software Engineering Conference*, pages 68–75. IEEE Computer Society, August 2001.
- [3] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proceedings of TACAS’01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Genova, Italy, April 2001.
- [4] S. Cohen. Jtrek. Compaq, <http://www.compaq.com/java/download/jtrek>.

- [5] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transaction on Software Engineering*, pages 1–22, March 1996.
- [6] J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
- [7] D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.
- [8] E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. New York Academic Press, 1968.
- [9] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.
- [10] J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.
- [11] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
- [12] K. Havelund and T. Pressburger. Model Checking Java Programs using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
- [13] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV’01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
- [14] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE’99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
- [15] E. Knapp. Deadlock Detection in Distributed Database Systems. *ACM Computing Surveys*, pages 303–328, Dec. 1987.
- [16] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison-Wesley, 1997.
- [17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
- [18] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, pages 37–48, Nov. 1989.
- [19] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
- [20] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE’00: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.